Analyzing the Evolution of Software by Change Analysis

Egon Wuchner
egon.wuchner@siemens.com
Jürgen Salecker
juergen.salecker@siemens.com
Siemens AG, CT T DE IT1, Munich, Germany

Software Maintenance Nightmare – Why does it happen?

Technical Debt
“Other debt accumulates from taking hundreds or thousands of small shortcuts—generic variable names, sparse comments, creating one class in a case where you should create two, not following coding conventions, and so on.

This kind of debt is like credit card debt. It’s easy to incur unintentionally, it adds up faster than you think, and it’s harder to track and manage after it has been incurred.”

Steve McConnell

Reasons
• “… debt are commonly incurred in response to the directive to “Get it out the door as quickly as possible.””
• Pressure due to time-to-market, lack of tests
• These costs due not show up at the beginning, defects show up later
• Because you need highly motivated and qualified engineers to put emphasis on quality
Software Maintenance Nightmare – Why does it happen?

Technical Debt
"All successful software gets changed. So if we think we’re working on code that will be successful … we need to keep it easy to change. Anything that makes code difficult to change is technical debt.

Just like any other debt, the cost of paying off technical debt gets more and more expensive over time. … Technical debt drives the total cost of software ownership relentlessly higher … eventually we will have to pay it off or the system will go bankrupt."

Mary Poppendieck

Financial Debt
• Incurs interest payments

Static tests
• Coding guidelines, programming standards, code inspections => manuals checks

• Code analysis tools
  aim at high code quality and less defects
  => automated checks
  => reduce manual inspection effort

• Architecture analysis tools
  aim at a maintainable and testable dependency structure

Dynamic tests
• Test, test and test – unit test, integration tests, system tests, acceptance tests
• Test-driven development

Agile methodologies, e.g. Scrum
• efficient and necessary roles; Customer involvement
  • small iterations, commitment – we fail or succeed together
  • fast feedback: sprint reviews, retrospectives
Software Maintenance Nightmare –
Architecture analysis tools

Usage scenarios
- Search dependencies, get overviews
- Monitor and evaluate dependencies for a hierarchical and testable design
- Identify cyclic dependencies
- Simulate refactorings by what-if analysis
- Take analysis results to generate restructured artifacts

Tools (all commercial)
- Understand
- Sonargraph/Sotoarc
- Lattix
- NDepend
- (Re)Structure101

Software Maintenance Nightmare –
Emergency case scenarios

No high test coverage

No code analysis, architecture analysis tool has been used

You have done this kind of quality assurance, but the size of your systems implies some technical debt to take anyway.

Results of analysis tools give you hints about a lot of pain points. Even starting with the main ones implies some initial effort.

Nevertheless,

You would like to understand how the code has evolved.

You would like to know why the code has evolved the way it did.

You would like to know where to look at else when you change code.
Goal: Increase the level of comprehension by analyzing changes.

- Stakeholders need to understand changes beyond text diffs
- Each person is focussed on a fraction of work, it is hard to perceive relevant changes beyond ones perception.
- It is especially hard in case of multiple teams or multiple locations.
- Stakeholders need to understand changes better: why has this part of the code changed

Koskinen, University of Jyväskylä, Finland
http://users.jyu.fi/~koskinen/smcosts.htm

85% of costs are devoted to system maintenance & evolution

50% of the time is spent in the process of understanding the code
Change Analysis –
Why are changes hard to recognize?
Change Analysis –
What about these changes?

Page 11

Change Analysis –
What about these changes?

Page 12
Change Analysis –
Why is it hard to recognize certain changes?

Because we lack abstraction w.r.t changes

But, we can start with

- well-known design categories: components, required/provided interfaces, dependencies
- change categories: added/changed/removed
- parameterized annotations: for design principles
  - e.g. annotation for error handling with parameters like PropagateUp, TranslateTo, NotifyUser, FailGracefullyByReturningDefaultValue, LogError

Change Analysis –
Levels of change abstraction

Structural levels of changes
- Changes on the level of the architecture, design, code, tests, requirements
  - Architecture components (artifacts, directories)
  - Design constructs (functions, class methods)
- Relation of changes on different levels of abstraction
- Dependency changes
  - Calls, called-by for design constructs
  - Provided and required interfaces for architecture components (artifacts, directories)

Change categories
- added, removed components
- changed provided component interfaces
- changed required component interfaces
- changed component implementations
- added, removed, moved, renamed, similar to, changed design constructs
Change Analysis –
Hot spots

What has changed the most over a period of time?

Hot spots granularity
- Architecture components, design constructs
- Traversing between component and design hot spots
- Aggregating the numbers up the hierarchy

Hot Spot properties
- Last commit
- Number of commits
- Percentage of changes
- Number of lines of total code

Change Analysis –
Rationale of changes

Relate requirements, defects to design and code to improve impact analysis of new changes

- which parts of an artifact, design construct contribute to a requirement (e.g. hazard key) or bug-fix?
- which set of architecture components, design constructs and code lines contribute to a requirement or bug-fix?
- whom to ask about the rationale of a specific change?
Future Change Analysis – Reverse engineer the rationale of changes

Select a subset of changes (e.g. over the last weeks) and relate them manually to an issue

- This feature could be part of the Change Analysis functionality

Reverse engineer the design and architecture

- Analyse design constructs and code used in common by many requirements

Relate requirements/use cases/scenarios/tests to the current code base automatically

- Take coverage tools and their output as input to a change analysis
- Take tracing tool output as run-time information input to a change analysis

Change Analysis – Maintenance Trend Analysis over the application lifecycle

How invasive has a requirement been implemented?
How local has a bug-fix been? Trends over time?

Feature invasiveness
- how many design construct have been changed for a feature implementation?
- invasiveness should not increase over time, otherwise there is certainly the need for refactoring.

Bug-fix locality
- how many design construct changes a bug-fix implies. How ‘local’ (focused instead of widespread) are these changes?
- the locality should become greater over time.
Future Change Analysis

Detect change (co-)relations over the application lifecycle

**Change prognosis**

- Which design constructs have often been changed in common?
- Which requirement implementations are (co-)related? On which level?
- Which changes can be clustered?

**Interaction of issues**

- Which requirements are affected by which bug-fixes? Bug-to-feature relation.
- Which manual tests are affected by a set of changes? Thus, which manual tests are recommended to do first?
- Which client software has to be build due to changed dependencies?

Application Lifecycle Features of Change Analysis

**Several change categories. One change category does not fit all.**

*Developers and architects understand changes on the respective level.*

**Understand the code in relation to issues,** (recover the rationale)

*Developers and architects get the relevant information for their impact analysis.*

**Change prognosis, (co-)relate changes, (co-)relate features**

*Stakeholders get guidance about possible side-impacts of changes.*

**Applicable to many scenarios**

- Development within a team, multiple teams, multiple development locations
- Developing up to the first release, Maintaining a system, new features, removing defects, adapting to new technologies
- Applicable to any level of testing or quality analysis tool usage or to any development process
## Application Lifecycle Features of Change Analysis

<table>
<thead>
<tr>
<th>Determine the feature Invasiveness</th>
<th>Relate bugs to features, hot spots</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developers/testers can assess the extent of verification&amp;validation effort.</td>
<td>Project managers can reliably spot pain points of development.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relate implementation changes to manual tests</th>
<th>Change prognosis of latest changes, Determine feature invasiveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration/System testers prioritize the large set of manual tests.</td>
<td>Project managers have quantifiable data to decide upon release inclusion.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Detect dependency changes</th>
<th>Determine maintenance trends</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build managers get hints which client systems to build only.</td>
<td>Project managers/R&amp;D managers have quantifiable data to know the amount of technical debt</td>
</tr>
</tbody>
</table>

## Future Change Analysis

### Change Analysis on Architecture and Design Models (e.g. UML, SDL)
- Using the [Atego Workbench](http://www.atego.com) for Application Lifecycle Management
- apply change analysis of the architecture and design on modeling level
- relate changes over different modeling types
- integrate model changes and manual code changes per time period

### Change Analysis over artifact types of the application lifecycle
- determine the volatility of requirements, models etc.
- relate changes over different artifact types and phases
- e.g. relate all changes of requirements, design and code for a change request
## Current status of the Change Analysis platform

<table>
<thead>
<tr>
<th>Feature</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Software pilot project at two business units</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Possibly going open-source after successful software pilots</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Adaptions to customer specific tool landscape</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Supposed to present experience report in 2013</strong></td>
<td></td>
</tr>
</tbody>
</table>
| **Multi-language support**                                  | - C/C++, C# (using Understand)  
- Python          |
| **Configuration Management support**                         | - Subversion, Git          |
| **Client/Server architecture, Browser-based client**        | - ongoing work on server scalability  
and server performance |
| **Platform independence**                                   | - implemented in Python  
- Interoperable with C# (IronPython.NET) |